Development of the System Assurance Reference Model for Generating Modular Assurance Cases

Andrzej Wardziński*, Aleksander Jarzębowicz*†
* *Argevide*, Gdańsk, Poland
{andrzej.wardzinski, aleksander.jarzebowicz}@argevide.com
† *Department of Software Engineering, Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology*, Gdańsk, Poland
olek@eti.pg.edu.pl, ORCID: 0000-0003-3181-4210

***Abstract*— Assurance cases are structured arguments used to demonstrate specific system properties such as safety or security. They are used in many industrial sectors including automotive, aviation and medical devices. Larger assurance cases are usually divided into modules to manage the complexity and distribute the work. Each of the modules is developed to address specific goals allocated to the specific objects i.e. components of the system's architecture. Such goals are applicable for given conditions of use, for instance, operational modes or target environments. It is recommended that the complete context of each of the argument modules, encompassing information about systems/components, goals and conditions of use, is described explicitly to enable efficient management and proper use of each module. This becomes even more important for component-based design, including the use of out-of-context components. In this paper, we describe a concept of a generic System Assurance Reference Model (SARM), which bridges the gap between assurance cases and the related context models. We identify the key factors that condition the high-level assurance case structure, explain how they can drive its decomposition into assurance case modules and outline the process of creating and using context models. We present a prototype solution which implements the SARM model and enables automatic data flow between models and assurance cases.**

Keywords—assurance case, safety case, argument templates, context models, contract-based design

I. Introduction

Assurances cases (ACs) are gaining increasing interest in industry, as their development and submission is required for high-assurance systems by a number of standards from several business domains, such as automotive [1][2], railway [3], aviation [4], autonomous systems [5] and medical devices [6]. An AC is a structured argument that justifies a claim about a specific property or properties of a considered system through an explicit chain of reasoning based on a body of evidence [7]. The top claim is supported by reasoning and the subclaims, which in turn need to be supported by further subclaims or evidence. The line of reasoning should be presented in an explicit way down to the level of supporting evidence. ACs are most commonly used to justify safety e.g. [1][3] (and as such are called safety assurance cases or simply safety cases), but other properties such as security [2] and dependability [8] are considered as well.

Despite such growing interest, the current state of practice is reported to be coping with several problems [9]-[11]. For example, an interview study with AC practitioners focused on challenges related to their work [9] revealed the most important problems to be those associated with the size and complexity of developed systems (and the corresponding ACs) and with change management. The main underlying factors behind such challenges according to [9] were: limited tool support (inadequate to practitioners' needs) and insufficient process integration between AC development and other project activities supported by other tools.

Contemporary systems usually consist of a large number of subsystems and components, organized in multiple levels of decomposition [12][13]. Moreover, several versions or variants of such system can be developed by substituting some components with others or adding/removing components responsible for specific features, which results in families of similar systems i.e., product lines [14]. AC development tasks are interrelated with other activities dedicated to the construction of the corresponding system and the overall system development lifecycle [11][15]. The influence between them is mutual. On the one hand, an AC uses (as evidence) the products of other activities e.g. system architecture, hazard list, test results. On the other hand, the work on AC development can result in changes regarding design decisions or the need to perform additional quality assurance actions. Consequently, there is a need for process integration to ensure that traceability is maintained, and changes are promptly reflected in all related artefacts. Attempts to adopt the Agile approach to high-assurance systems development [16][17] and recent ideas of continuous assurance [13][18] make this need even more crucial as shorter iterations and feedback loops require more rapid responses and more effective change management. In the case of Agile projects, automation is a necessary condition as they usually rely on DevOps and automated releases.

`

A number of AC-supporting software tools are available [19][20]. However, significant demand for advanced AC tools—especially with respect to scalability, modularization, automated generation, and traceability to external artefacts describing the AC context—is still reported [9][13][21].

Modularization is an effective way to cope with complexity [22] because it makes it possible to partition an AC into separate but interrelated modules. When dividing an AC into modules and working on them separately, there is a need to ensure the right context for each module. By context, we mean the conditions under which a given AC argument is valid. The argument module may not be applicable in a different context. The context of each module needs to be explicitly defined in order to ensure the module makes a valid composition with other modules and that it is correctly reused. This is especially crucial for product lines, as many similar (but not exactly the same) components are used and a proper AC argument has to be provided for each version of each component. The information that defines the context is managed in various supporting tools and it becomes essential to enable a simple, universal approach to automated data transfer between such tools.

We aim to provide a solution with the potential to address such concerns. To achieve this, in this paper, we identify the key factors that condition the high-level AC structure, explain how they can drive its decomposition into AC modules, and outline the process of creating and using context models. A main idea we introduce is the System Assurance Reference Model (SARM), which bridges the gap between ACs and the related context models. In this paper, we present:

1. A concept of a unified context model which covers the complete AC context: the object (system, component), the assurance goals and the operational conditions. The idea is not to develop the context models from scratch but to integrate with the existing data which describes the context relevant to the AC process.
2. The reference model which connects ACs with their context models on two levels: AC templates (reusable argument structures) with the abstract context model, and AC arguments with the concrete context model specific for a given system. The reference model supports modular ACs and enables two-way traceability.
3. A grammar for parameters and conditions in AC templates to control the references to the argument context in the template instantiation process.
4. The idea of context tree diagrams for the compact presentation of context dependencies in AC templates.
5. The algorithm of modular AC generation from modular templates with the use of context models.
6. Demonstration of the approach with the use of a prototype tool which implements argument generation.

The remainder of this paper is organized as follows. In Section II, we describe the background and in Section III, we outline the main related work. Section IV introduces the concept of SARM and is followed by Section V, which provides more details about SARM logical design. In Section VI, we present an illustrative example of SARM application. The paper is concluded in Section VII.

II. Background

A. Goal Structuring Notation

GSN (Goal Structuring Notation) [7] is the most popular notation for ACs and it will be used in this paper. Fig. 1. lists selected elements of GSN (the ones that will be used in the remainder of this paper).

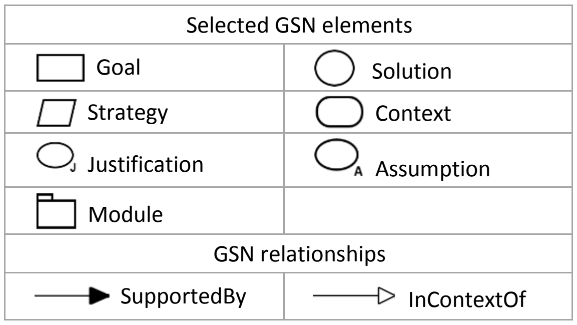

Fig. 1. Core GSN notation summary.

GSN assurance cases include Goals which express claims about system properties. Strategies are used to explain the argument step including a given Goal and its supporting elements. A Justification provides a rationale that the reasoning defined by a given Strategy is valid and complete. An Assumption is a statement that is not argued nor demonstrated by evidence, but simply treated as true. A Solution represents an evidence item. A Context represents the contextual information for which the AC is valid. The reasoning and evidence support is indicated by the SupportedBy relationship, while the InContextOf relationship links additional elements such as Context, Justification and Assumption. The elements and relationships defined in GSN make it possible to develop complex argument structures.

B. Methodological foundations

Several methodological solutions have been proposed for ACs to cope with their complexity, to limit the impact of changes and to reduce the work effort necessary for AC development and maintenance:

- Modules and contracts [22]-[24]: ACs can be partitioned into separate but interrelated modules. A module can be, e.g., dedicated to a particular component of the system or aimed at justifying a specific claim. The interrelationships between modules can be captured in contract modules. A contract module is placed between AC modules to ensure that one module provides valid support in the full context required by another module. The context of each AC module should be defined explicitly to use contract modules effectively. The correct management of the context of AC modules helps in effective argument encapsulation to make the AC architecture less prone to changes. It makes it feasible to change or replace

AC modules while maintaining the same context and without the need to change other modules.

- Patterns [25]-[27]: An AC pattern is an abstract reusable argumentation structure that captures a repeatedly used successful argument. As such, AC patterns are analogical to the design patterns known from the software engineering domain. A pattern includes parameters (e.g. the name of the mitigated hazard). Such parameters need to be assigned concrete values in order to use the pattern in a particular AC—this process is called pattern instantiation. Patterns can capture argumentation structures of different size and complexity, however, they are usually of a local nature, i.e. to be applied at a given step of argument decomposition.
- Templates [28]-[30]: An AC template is a partially complete AC structure that can be determined beforehand for the specific system under consideration. For example, it can capture the requirements of a given standard or provide the structure of an argument commonly used for a given class of systems. A concrete AC can be constructed based on a template by extending the template with concrete evidence and possibly with more specific argumentation. The real difference between a template and a pattern is one of completeness—a template is a complete AC, in which claims are specialized for the specific situation [29]. Similarly to patterns, templates can include parameters to be instantiated when constructing a concrete AC.

The effective use of such solutions is dependent on the availability of appropriate tools which support modularization, automated pattern instantiation (and maintenance), consistency control, change management and traceability. This can be achieved through integration with other tools used in the system life cycle to use available information relevant for ACs. The approach where ACs are constructed and maintained in a tool-supported, automated manner on the basis of information included in external artefacts (usually models) is called Model-Based Assurance Case (MBAC) and is one of the most promising directions in the AC domain [31].

III. Related Work

Solutions enabling the automation of AC development and maintenance are a relatively recent achievement. The research in this domain includes the following areas:

- Context models and traceability mechanisms between AC fragments and external artefacts and their consisting elements;
- Automated development and/or maintenance of ACs through generation or update of their fragments based on external models (for example, a hazard model or a threat model).
- Coexistence of automatically generated and manually managed arguments.

ACs dedicated to the safety properties of a system are strongly dependent on hazard analysis uncovering the dangerous situations that could lead to accidents. As a result, a number of approaches aimed at partial generation of AC fragments dedicated to hazard mitigation have been proposed, e.g. [32]-[34].

Other essential artefacts guiding AC development include various system models (describing the considered system). Architecture Analysis & Design Language (AADL) models developed in tools such as OSATE [35] or AutoFocus [36][37], as well as multiple other models [38], were used as a basis for automated generation. Also, architectural design patterns expressed in Unified Modeling Language (UML) can drive AC fragment generation [39].

Apart from the arguments related to the considered system itself, AC often also covers so-called process aspects, i.e. argues that the process of system development follows the requirements of standards and/or good practices. Thus, attempts have been made to model the development process and to use such models to generate the corresponding fragments of ACs [40]-[42]. Other possible artefacts driving automation include safety contracts between system components [14][43] and the results of formal code analysis provided by another tool [44].

The individual artefacts and information sources mentioned above can also be combined to provide a more comprehensive basis for AC generation. A quite widely explored path covers the combination of system architecture and information about hazards and/or safety requirements [45]-[48]. Another possibility is the use of both system and process models to include product-based as well as process-based aspects in an AC [49]. A more generic approach, where an AC can be designed using an artefact tree consisting of various external artefacts and definitions of relations between them is also possible [50]. A proposal dedicated to ACs for machine learning models, using Operational Design Domain (ODD) models and training datasets as artifacts driving automation, can be found in [51].

Despite the availability of methodological approaches and tools, the foundations of MBAC, such as a formal or semi-formal definition of parameters or an algorithm responsible for the instantiation process, are rarely described in detail. Formalized definitions of AC patterns and their parameters are presented in [45][52][53]. Less formal ones, based on the Structured Assurance Case Metamodel (SACM) metamodel [31] or Extensible Markup Language (XML) [47][48], can also be found. The algorithms defining the instantiation process are published (as pseudocode or an algorithm based on formal definitions) in [26][38][40][43][46][53].

Our proposal differs from the related works presented above by the following distinguishing features:

- The other works focus on system models, goals (requirements or addressed hazards) or a combination of both. The environment, including different operational situations/modes as well as environmental conditions, is however not explicitly covered, while such aspects may be crucial for many systems [54]. At the same time, the importance of the context is increasingly recognized—for example, the concept of Operational Design Domain (ODD) was identified and included in recent standards for autonomous driving in automotive [55][56]. Only [51] deals with ODD, but for a very specific use case of arguing the safety of machine learning models applied to automated driving systems. Our approach is therefore a more general one, allowing the system to describe various elements of context in a uniform way and to include a broader scope of external artefacts.

- The existing proposals, except [14], aim at generating particular AC fragments dedicated to, e.g., a given hazard, requirement or system component. We focus in this paper on high-level AC decomposition into modules and providing initial templates for each module to enable its further development.

IV. Concept of System Assurance Reference Model

Our goal is to manage the AC context in an explicit and systematic way with the use of a general context model. As we could not find any solution that was ready to use, we decided to develop a System Assurance Reference Model (SARM) to address the problem. Our main goals and requirements for the models and the process of their use can be summarized as follows:

- The solution should provide a uniform way to manage references to any type of context models relevant to ACs. It cannot be limited to a specific subset of models with respect to their purpose or type (e.g. system models, hazard models, requirements models, process models).
- The models should operate at two levels: the abstract level of argument templates, which are applicable to specific types and classes of systems, and the concrete level of argument modules for specific systems and devices.
- Traceability in both ways should be provided between the abstract and concrete levels of the model. The solution should be capable of answering questions such as which AC modules apply for a specific device.
- The solution should support reuse to the greatest extent possible with regard to both ACs (e.g. an AC template that can be instantiated for different systems or components) and context models (e.g. a given model is referenced in different parts of an AC).
- Scalability has to be provided. The number and size of the context models (e.g. for product lines and/or systems built from a large number of separate, out-of-context components) can be substantial, but that cannot limit the application of the intended solution.
- The approach should support the evolution of all models and maintain consistency. No restrictions of independent changes in context models can be accepted. Also, argument templates may evolve and be modified. The approach should support the evolution process, provide for traceability of all changes and control the consistency of all interrelated models.
- The approach should allow the use of any data format for source context models. When necessary, a method of model transformation should be provided.

Our further considerations will be based on the following definition of an AC, which is a slightly modified variant of existing definitions (e.g. [7]). In the definition, we have highlighted the sections relevant to our goals.

An assurance case provides confidence that the described ***system*** *(object of the argument) guarantees satisfaction of the* ***assurance requirements*** *under the assumptions of the specified* ***context of use****.*

AC top claims are usually defined with three main types of context information: (1) the system (the object of the argument), (2) assurance goals (e.g. safety requirement, assigned Safety Integrity Level, regulations to comply with) and (3) the environment and conditions of use (e.g. flight phase, weather conditions). Each AC argument can be understood as a statement and justification of some guarantees (assurance goals), provided that some assumptions (context of use) are satisfied. For example, a medical device is safe in operation, provided that it is used by trained clinicians in an intensive care hospital environment.

Each type of context information may be used for AC decomposition into modules. A user may decide to decompose an AC into modules according to assurance goals, such as safety, security and conformance to specific standards. Another approach is to develop AC modules for a system, its subsystems and components.

The model describing the system conditions of use can also be used in the argument decomposition. Separate argument modules or branches can be developed for different conditions of use, e.g., military and civilian missions, vehicle operation in populated or unpopulated areas, or in summer and winter conditions.

Context models for ACs usually cover system architecture, the risk model and assurance goals, the environment and conditions of use. From the perspective of a person responsible for AC development there is a need for a generic and uniform way of referring to the context elements in the argument.

We assume that each element of the context model can be referred to with an URL address. On the AC side we use the tool that provides unique URL address for each argument module and element. This will enable tracing dependencies in both directions.

It is important to emphasize that our objective is not to propose any new context models or a new notation to express them, but to track the relations to the models used in industrial practice in order to manage ACs in a more efficient way.

Our proposed solution is called SARM (System Assurance Reference Model). The main components of SARM are shown in Fig. 2. The arrows in the diagram indicate that the given component is based on another one. An arrow from A to B means that B is defined using elements (terms) from A. In practice, it is implemented with references (links) from elements of B to elements of A.

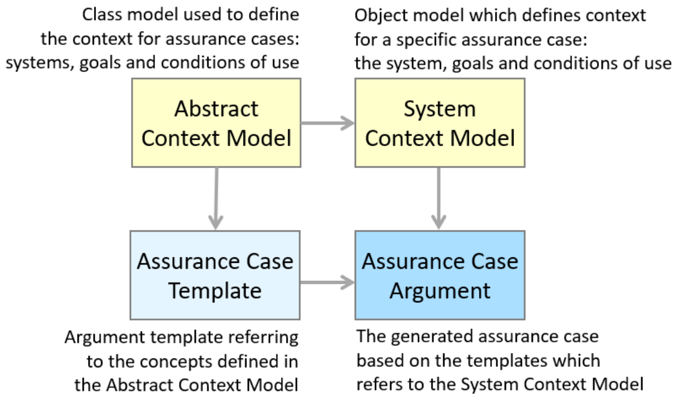
Class model used to define the context for assurance cases: systems, goals and conditions of use
Object model which defines context for a specific assurance case: the system, goals and conditions of use
Abstract Context Model
System Context Model
Assurance Case Template
Assurance Case Argument
Argument template referring to the concepts defined in the Abstract Context Model
The generated assurance case based on the templates which refers to the System Context Model

Fig. 2. Main components of SARM.

Abstract Context Model (ACM) defines the terms of the context model to be used in the AC templates and, consequently, in the AC arguments. The terms have to cover the object-conditions-goals triad mentioned earlier. It is not our intent to create a complete model covering the needs of any AC application, therefore, the terms defined in such a model will differ for different applications. For example, the context model can include a system's physical or functional architecture. As for assurance goals, the context of a safety case can include the risk model: the terms of hazards, their causes and safety requirements, while a security case would be based on terms such as threats, vulnerabilities, etc.

Assurance Case Template (ACT) is an argument template (the explanation of AC templates is provided in Section II), which refers to the terms defined in a specific ACM. For example, a template for arguing a system's safety by addressing hazards with safety functions assigned to particular subsystems would require references to the terms "system", "subsystem", "hazard" and "safety function" defined in the corresponding ACM. The template can be divided into a set of modules. An example is presented in Fig. 3.

System Context Model (SCM) describes the context for a particular system including information about the assurance goals and conditions of use. The model should use the terms defined in the ACM. For example, the architectural decomposition of a considered system into subsystems can be described using the model of system decomposition into subsystems: the Autonomous Vessel X (system) is composed of the Situation Awareness (subsystem) and the Control Execution (subsystem).

Assurance Case Argument (ACA) is based on a template (ACT) and on a system context model (SCM). The argument (ACA) is a result of the templates' (ACT) instantiation using the context information (SCM). The SCM is used to fill the ACA with specific content, including, e.g., the names of the subsystems, safety functions implemented by them, hazards addressed by those safety functions, etc. When the template is divided into modules, the instantiated AC architecture will follow the same decomposition.

The SARM model is accompanied by the process of its development and maintenance. The process is outlined in Fig. 4 and consists of the steps explained as follows.

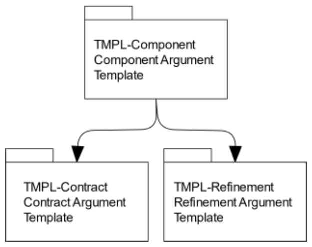
TMPL-Component
Component Argument Template
TMPL-Contract
Contract Argument Template
TMPL-Refinement
Refinement Argument Template

Fig. 3. Example of a hierarchy of AC modules.

A. *Step 1*

Step 1 is dedicated to the abstract models that provide the basis for the models created in the subsequent steps. The objectives of Step 1 are to: (1) Develop an ACM, which should define the terms relevant to ACs for a given domain and objective (e.g. class of systems, conformance to a specific standard); (2) Develop an ACT (i.e. a template) using the terms defined and included in the ACM. We describe these activities together because we consider them to be closely related. Even though the ACM is a more "basic" model, its contents can be influenced by the needs identified during the template development. When a new term to describe argument modules or branches in the template is needed, it is necessary to revert to the ACM definition and to extend the ACM with the required properties. We recommend the strategy of defining the minimum required scope of the ACM. The context models should be kept as simple as possible and contain only the information needed in the ACs. Usually, this will be a small subset of the complex models used in system development.

B. *Step 2*

In this step, an SCM is created to describe a specific system. It defines a set of objects being the instances of classes defined in the ACM. Each element of the system context should be described as an object following the abstract context model (ACM). The system description should provide a complete set of information for the template instantiation.

C. *Step 3*

The goal of this step is to develop an assurance case argument (ACA). This step consists of both automated and manual parts. The automated action is the generation of the ACA based on the template (ACT) with the use of context information (SCM). This is the first action, as it creates the AC modules that can then be supplemented by manual actions.

The manual part is the development of low-level argument sections and providing the evidence. The manual part should not interfere with the automatically generated argument parts to maintain the consistency with the context models.

In this paper, we mainly focus on the automatic generation of modular arguments. The aspects of coexisting automatic and manual argument development and maintenance are beyond the scope of this paper and are not discussed in detail. Our basic approach is that the users should extend the generated arguments, but not modify the automatically generated content.

D. Step 4

Step 4 covers the maintenance of the assurance case argument (ACA). This includes changes in any of the other AC modules. There are generally two main types of model changes. The first and the most common type is the change in the system context. The system architecture can be modified, or its goals may be refined, or the conditions of use may evolve. Any of these changes (in the SCM) may require an update of the argument (ACA). The second type is the changes in the templates. Sometimes they may also imply changes to the context model ACM. In some cases, they may also require extending the system description in the SCM.

The evolution of SARM models should ensure the consistency is maintained between the related models. We are aware of the importance of this step, however, we will not address maintenance issues in this paper.

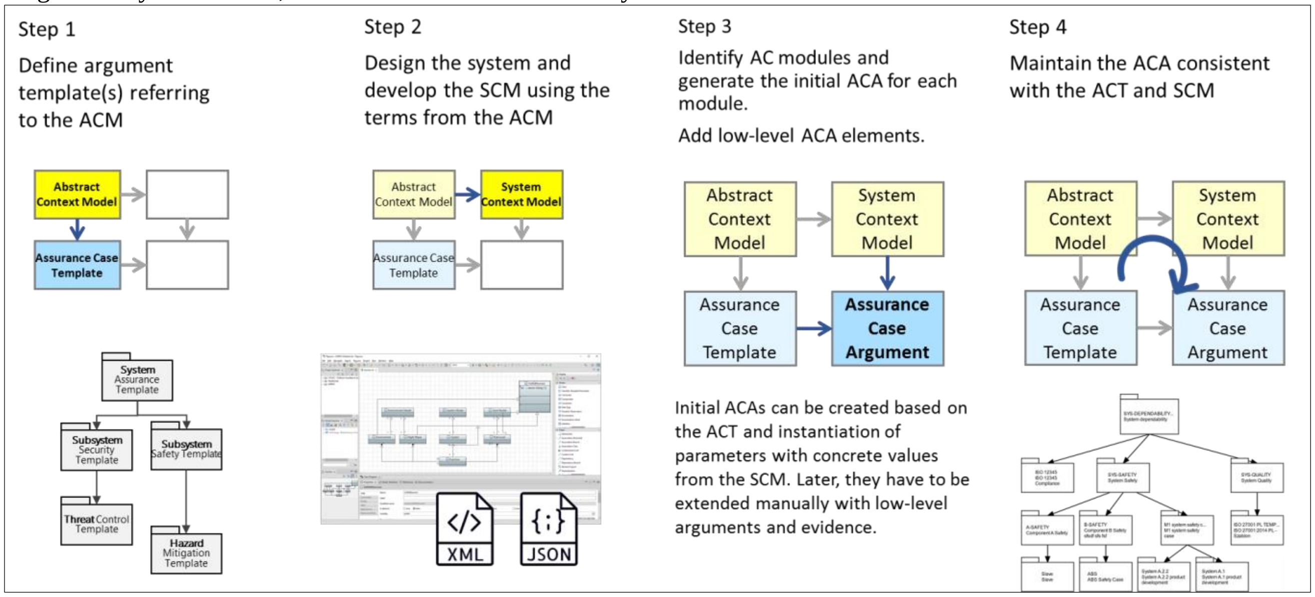
Step 1
Define argument template(s) referring to the ACM
Abstract Context Model
Assurance Case Template
System Assurance Template
Subsystem Security Template
Subsystem Safety Template
Threat Control Template
Hazard Mitigation Template
Step 2
Design the system and develop the SCM using the terms from the ACM
Abstract Context Model
System Context Model
Assurance Case Template
XML
JSON
Step 3
Identify AC modules and generate the initial ACA for each module.
Add low-level ACA elements.
Abstract Context Model
System Context Model
Assurance Case Template
Assurance Case Argument
Initial ACAs can be created based on the ACT and instantiation of parameters with concrete values from the SCM. Later, they have to be extended manually with low-level arguments and evidence.
Step 4
Maintain the ACA consistent with the ACT and SCM
Abstract Context Model
System Context Model
Assurance Case Template
Assurance Case Argument

Fig. 4. Process of SARM definition and use.

V. SARM LOGICAL DESIGN

In this section, we will describe the models used to implement the concept introduced in Section IV and present the algorithm for automatic AC generation based on the templates (ACT) and system context model (SCM).

We designed a SARM as an extension of the argument model we used in the NOR-STA tool [57]. As we represent the perspective of NOR-STA's manufacturer and provider, we intend to implement the SARM as part of this tool, but currently, we are still exploring prototype solutions. Our goal is to keep the model used to perform the process described in the previous section as simple as possible. There is no decision yet whether the model should be based on the SACM metamodel [58]. The Artifact Metamodel would probably be useful to define the context information for the ACs, but at this stage of our work, we have not decided yet.

The SARM model consists of four parts and we will discuss them in the order they were developed.

A. Abstract Context Model (ACM)

The purpose of the ACM is to be referred to in the ACT, and this model should be developed first. The ACM model should make it possible to define any context models relevant to the ACs, so we will keep it as generic as possible. One could say that this should be a simple ontology. We followed this direction and defined two classes to allow the declaration of context objects and their properties.

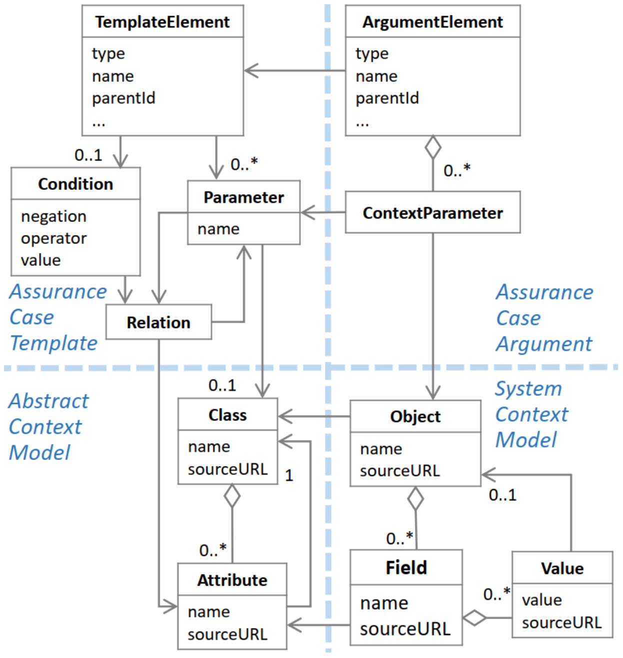

Fig. 5. Class diagram of System Assurance Reference Model (SARM).

`

The first one is used to define classes of objects (its name is Class, with a capital “C” in its name). The second class defines the Attributes. An Attribute can refer to any Class defined in the ACM or to a predefined Class which implements a basic data type such as “integer” or “string”. When the type of an Attribute is another Class defined in the model, we allow multiple values, i.e. a list of objects can be used as the value in the model implementation.

The ACM model consists of two classes only, but it is quite powerful and makes it possible to define any context objects and the relations between them. The model is presented in the lower left part of the diagram in Fig. 5.

B. *Assurance Case Template (ACT)*

We understand an argument template to be an abstract argument structure which contains a top claim and an almost complete argument. The result of a template instantiation is an argument module which may not be complete, in particular the appropriate evidence for the module context is still missing and should be provided.

The templates will refer to the ACM and we will use two types of references, which correspond to the two forms of abstraction defined in section 2:8.1 of the GSN Community Standard version 3 [7]. The parameters and conditions described below are our extension of GSN patterns, introduced for the purpose of formalization of model references:

- **Parameters** are used to specify references in argument templates. This implements the element abstraction defined in the GSN Standard. An example is the parameter “{Component C}” presented in Fig. 6. Parameters are used in ACT elements to indicate the need for their instantiation for each specified context entity. Parameters can also include dependencies. The parameter “{Contract K | C.contracts}” says that the element of the template should be instantiated for every contract K of the given component C.

- **Conditions** are used to manage conditional branches of the argument. This corresponds to the structural abstraction in the GSN Standard. Attributes of the context objects can be used to define conditions. For example (see Fig. 6), the argument branch with the condition “[C.subcomponents]” will be implemented only when component C has some subcomponents. The other argument branch will have a condition with an exclamation mark to denote negation—“[! C.subcomponents]”. This template element will be instantiated only when component C does not have any subcomponents.

The described mechanism of conditions in the SARM differs from the structural abstraction defined in the GSN. The GSN choice element (section 1:3.2.3 in [7]) makes it possible to define the relation ‘N of M’ for a group of argument elements. The problem with the automatic argument generation is that the criteria under which N elements out of M should be selected is not clearly defined. Instead of the ‘N of M’ choice operator, we propose precise definitions of the conditions specific to each template element.

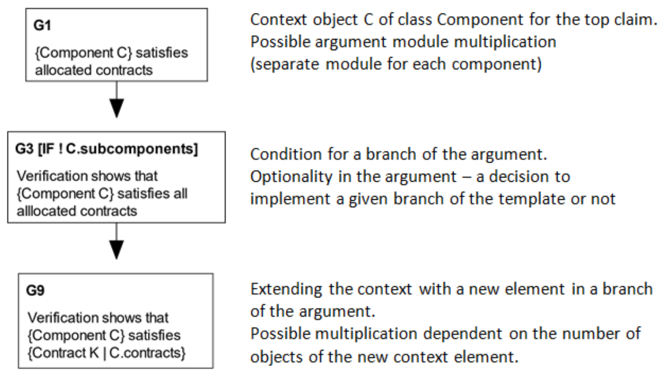
G1
{Component C} satisfies allocated contracts
Context object C of class Component for the top claim.
Possible argument module multiplication
(separate module for each component)
G3 [IF ! C.subcomponents]
Verification shows that {Component C} satisfies all alllocated contracts
Condition for a branch of the argument.
Optionality in the argument – a decision to implement a given branch of the template or not
G9
Verification shows that {Component C} satisfies {Contract K | C.contracts}
Extending the context with a new element in a branch of the argument.
Possible multiplication dependent on the number of objects of the new context element.

Fig. 6. Claims in AC template with parameters and conditions.

The parameters and conditions are presented in the template elements in brackets and they follow the syntax:

Parameter := <class> <object> [<object>.<attribute>]

Condition := [<negation>] <object>.<attribute> [<operator> <value>]

The (simplified) ACT model is presented in the upper left part of the SARM diagram (Fig. 5). It contains classes to describe the parameters and conditions as an extension of the template argument model. This is presented with the TemplateElement class, and we do not present any other classes used in the ACT. In particular, the relations between the template elements and interfaces are not presented in the model.

The context objects set for a specific argument element are inherited to all its child elements. To avoid potential conflicts of the context objects, we assume that the template has a tree structure. This assumption is implemented with the parentId attribute in the TemplateArgument class.

C. *System Context Model (SCM)*

SCM describes a concrete context model which is to be used for a specific AC. It is shown in the lower right part of the SARM diagram (Fig. 5). The model includes instances of ACM classes that describe a concrete system, its goals, and conditions of use. The method of creating SCM objects depends on the system design process and the tools used. If some modeling methods are used for a system (UML, AADL, etc.), an SCM could possibly be extracted from these models. Another source of the model data can be the Operational Design Domain (ODD). Finally, some context data cannot be formalized and may require a dedicated solution.

Regardless of the method of development of the SCM, it should be consistent with the abstract model (ACM). If there are difficulties defining the SCM based on the existing model structure, a return to the first step should take place (to extend the ACM with the required new classes or attributes).

The relationship between the abstract and concrete model for a specific system is illustrated in Fig. 7. All entities in the SCM should refer to their ACM counterparts.

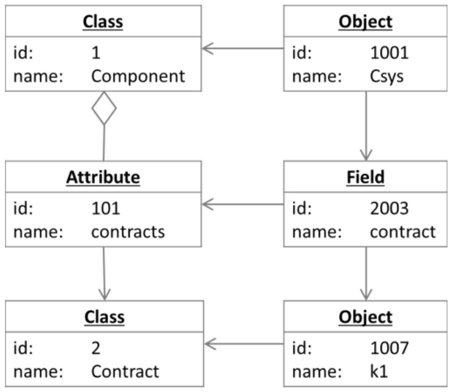

Fig. 7. Relations between objects of the system (on the right) and the abstract (on the left) context model.

D. Assurance Case Argument (ACA)

The argument (ACA) modules are to be generated from the ACTs based on the context model (SCM). They should maintain relations to both the ACT and SCM. This model is presented in the upper right part of the SARM diagram (Fig. 5). The class ArgumentElement denotes the base argument element (Goal, Strategy, etc.) and further details of it are not presented. The relation to the SCM is an extension of the base model specific for the SARM.

The ACA is a model of modular arguments that is to be generated from other models. We have developed an algorithm to generate the arguments, which is presented in Fig. 8. The argument generation is performed in three steps (described in the algorithm).

1. Identification of all possible modules for the provided templates and the system models.
2. Generation of argument bodies for each module.
3. Connecting the argument modules with interfaces

VI. Illustrative example

Generating modular ACs using SARM models will be demonstrated using an illustrative example with a small system consisting of four components. The example will involve the use of a modular template which contains references to the SCM.

In the example, we will use our prototype SARMER tool. The tool supports generating modular arguments according to the algorithm presented in Section V using SARM models. The JavaScript Object Notation (JSON) data format is used for all models to provide easy integration with other tools. The input data consists of abstract and system context models (ACM and SCM), and an assurance case template (ACT). The tool performs an analysis of the input models and produces the model of a modular assurance case argument (ACA) in JSON format. Additionally, in the last step, SARMER produces an XML file to import the generated modular ACA into NOR-STA.

```
GenerateACA(ACM, ACT, SCM)
  Create argument modules
  Generate bodies of argument modules
  Bind modules with interfaces

Step 1. Create argument modules
  For each template
    get the top claim and its context parameters
    find all possible objects which satisfy the context requirements
    for each possible set of objects
        create an argument module
        add top claim and set its context objects

Step 2. Generate bodies of argument modules
  For each argument module
    get the top claim and its context
    ExpandElement(top-claim)

  Procedure ExpandElement(element)
    find a corresponding templateElement for the element
    if templateElement has no children
      then return
    for each child of templateElement
      CreateChildren(element, child)

  Procedure CreateChildren(element, templateElement)
    if templateElement contains a condition
      if condition is not satisfied
        then return
    if templateElement does not contain any parameter
      or there are no new parameters
      then  create element from templateElement under element
            with the same context inherited
            ExpandElement (createdElement)
            return
    identify new parameters in the templateElement
    search objects for the new parameter(s) in the current context
       for each new object found
          create new context = the current context and the new object
          create element from templateElement under element
                with the new context
          ExpandElement (createdElement)

Step 3. Bind modules with interfaces
  For each argument module
    for each element E in the module with a required interface
      find a corresponding templateElement
      read the interface of the templateElement
      find a supporting template
      for each argument module A based on the supporting template
        if E.context is the same as the context of the top claim in A
           bind E to the top claim in A
```

Fig. 8. Algorithm description.

The prototype tool has some limitations, including a simplified user interface and error reporting. Another simplification concerns the types of conditions implemented in the templates. Currently, the only implemented conditions are the checks if a given attribute value is null or not. The tool implements AC generation, but the functions to update the arguments when the context models change have not been implemented yet.

`

We have selected a simple component-based system for our example. The goal of the argument will be to demonstrate that a given component satisfies its assurance-guarantee (AG) contract. The first sample system we analyze is described in [14]. This is the Csys component, which is composed of three subcomponents: C1, C2 and C3. Assurance-guarantee (AG) contracts are used to describe the behavior of each component. The structure of the system is presented in Fig. 9. Symbols **A** and **G** denote the assumptions and guarantees of specific components. The arrows with **k** labels present contracts. An assume-guarantee contract is a property, meaning that a given component ensures that if the assumptions A are satisfied, the guarantees G are also satisfied. A simple example is a contract for a lamp: the assumptions are “the switch is in the on position” and “power is on”, and the guarantee is “the lamp is lit”. Some contracts are relevant for safety and ACs are developed to demonstrate that the component correctly implements the assigned contracts.

Demonstrating that the components satisfy their contracts is not sufficient. A composition of contracts of components should follow specific rules. For example, it should be demonstrated that a guarantee of one component is sufficient to satisfy an assumption of another component. These relations are called refinements. They are denoted with **r** labels in the diagram in Fig. 9. For more information on how AG contracts work and for sample system details, please refer to [14].

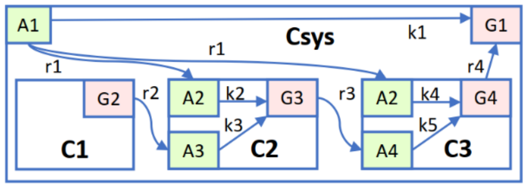

Fig. 9. System model used in the example.

The design of a composite component is considered correct when its contract is implemented with a continuous sequence of refinements and contracts of subcomponents. Contract k1 of component Csys in Fig. 9 is implemented by the sequence (((((r2, k3), (r1, k2)), r3, k5), (r1, k4)), r4). The implementation of k1 is complete and there are no redundant relations.

The example will be presented in three steps according to steps 1–3 of the process described in Section IV. The first step is to define the ACM and ACT. While it is possible to develop large and complex ACMs, our recommendation is to develop the general ACT structure first and then see what references are needed in it. The ACM should fit into the scope of the information needed for the ACT.

We used the NOR-STA tool to develop the AC templates (one is presented in Fig. 10) based on the schema described in [14]:

- For a composite component, we need to demonstrate that:
  - the component composition is correct, and
  - every refinement in a given component is satisfied, and
  - every subcomponent satisfies its contracts.
- For a primary component we need to demonstrate that:
  - every allocated contract is satisfied.

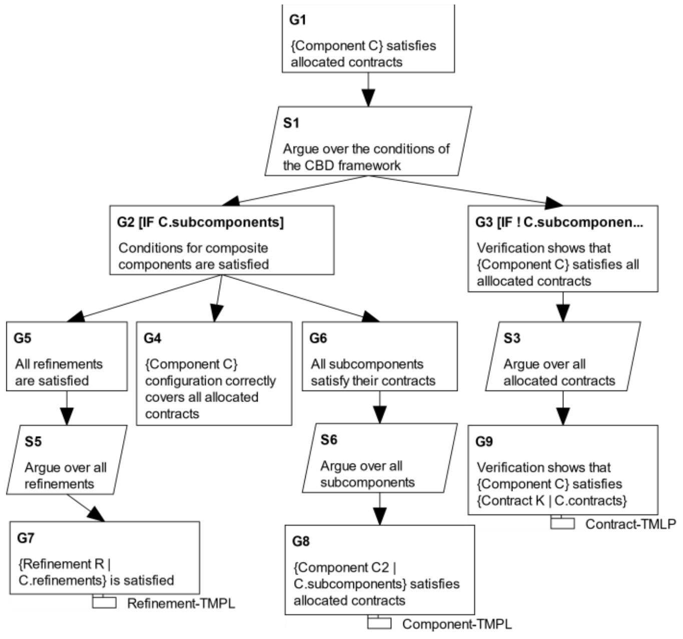
G1
{Component C} satisfies allocated contracts
S1
Argue over the conditions of the CBD framework
G2 [IF C.subcomponents]
Conditions for composite components are satisfied
G3 [IF ! C.subcomponen...
Verification shows that {Component C} satisfies all alllocated contracts
G5
All refinements are satisfied
G4
{Component C} configuration correctly covers all allocated contracts
G6
All subcomponents satisfy their contracts
S3
Argue over all allocated contracts
S5
Argue over all refinements
S6
Argue over all subcomponents
G9
Verification shows that {Component C} satisfies {Contract K | C.contracts}
Contract-TMLP
G7
{Refinement R | C.refinements} is satisfied
Refinement-TMPL
G8
{Component C2 | C.subcomponents} satisfies allocated contracts
Component-TMPL

Fig. 10. Assurance case template for a component.

The main concepts that we need to define in the ACM are: a component, a contract and a refinement. We will also need references to assumptions and guarantees in the template for a contract argument. Assumptions and guarantees are defined as specifications. Each specification has a ‘type’ attribute to indicate if it is an assumption or a guarantee. The ACM is presented in Fig. 11. It will be used for references in the ACT.

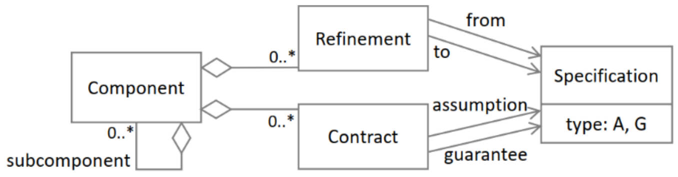

Fig. 11. Abstract context model (ACM) for component-based architecture.

In our example, we use the JSON format to represent the model. This data format is quite easy for humans to operate. We prepared the data for this example manually, but it is planned to be automated in the further phases of our work. In Fig. 12, we present a simplified fragment of the JSON file we used. We removed some parts from the JSON sample (e.g. identifiers) to make the code sample smaller for this paper. It presents the two main classes used in the ACM.

- A Contract class with two attributes: an assumption and a guarantee.
- A Component class, where a component can contain subcomponents (objects of class Component) and contracts (objects of class Contract).

`

```
{ "classes": [{
    "name": "contract",
    "attributes": [
      { "name": "assumption", "type": "specification" },
      { "name": "guarantee", "type": "specification" }]
  }, {
    "name": "component",
    "attributes": [
      { "name": "subcomponents", "type": "component" },
      { "name": "contracts", "type": "contract" },
      ...
```

Fig. 12. Excerpt of ACM definition in JSON format.

The ACM is referred to from the ACT. The ACT contains three modules, one of which is presented in Fig. 10. The templates include parameters and conditions referring to the classes defined in the ACM. The data describing the parameters and conditions in the template can be extracted and presented in the form of a context tree, as presented in Fig. 13. A context tree presents a clear view of the complete argument context at an abstract level. Finding all dependencies in larger arguments takes time and this type of diagram speeds up this process. We find this diagram to be useful for tracking argument dependencies.

The two abstract models—for the context references (ACM) and for the template (ACT)—are now complete. The next steps are performed for a specific system. The system we analyze is presented in Fig. 9 at the beginning of this section.

The SCM is developed using the JSON format. The model includes objects of classes defined in the ACM. This is quite a simple process, but we had to make one correction to the system structure presented in Fig. 9. The component C1 has a guarantee G2 and no contract inside it. The argument template requires a component to have contracts or subcomponents and C1 does not satisfy this requirement. We extended the system model with an additional contract, k6, with a null assumption and the guarantee G2.

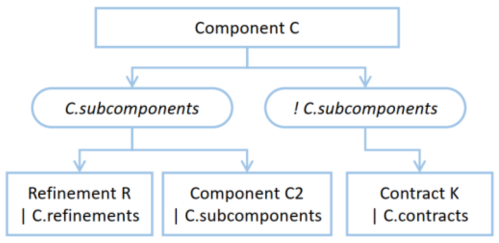

Fig. 13. Context tree of the ACT for a component.

The completed models provide sufficient data to generate the ACA for the specified system using the specified templates. This ACT consists of three modules and the result of argument generation is a set of interconnected argument modules of three types: for components, contracts and refinements. The modular AC presented in Fig. 14 consists of 14 modules: four modules for components, six modules for contracts (including the k6 contract described above) and four refinement relations.

The SARMER tool produces a single XML output file which defines the generated AC modules. The file can be manually imported into the NOR-STA tool. It is planned to automate this step using the NOR-STA API interface in the future. Diagrams like in Fig. 14 can be generated in NOR-STA when the ACA model is imported into the tool.

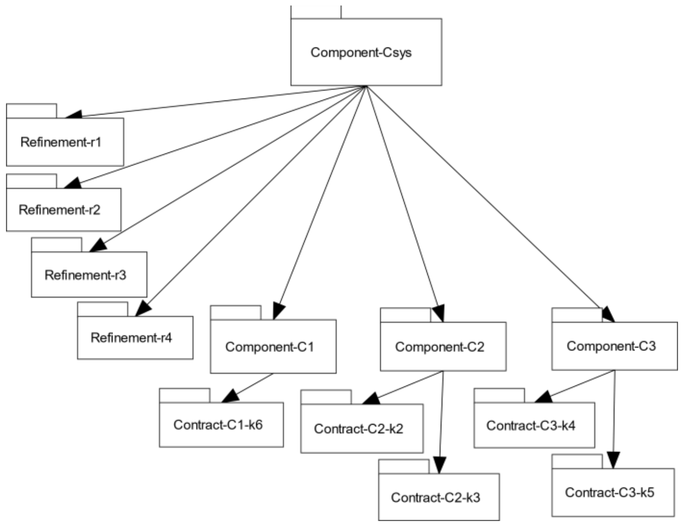

Fig. 14. Modular diagram of the generated assurance case.

VII. Conclusions and future work

Automation is practically a necessity for the management of large-scale ACs. We have presented a generic approach of AC architecture management based on context models. This reduces the need to manually develop large parts of AC arguments and it is expected that it will reduce the number of mistakes and problems with the arguments' correctness.

The System Assurance Reference Model (SARM), used to control the assurance case generation, can cover a broad scope of information including system architecture, assurance goals, risk models, environment and system life cycle activities. The reference model can be used at both levels of the abstraction: templates and concrete arguments. The reference model enables traceability of relations in both directions, to and from the context model. We have presented the logical design of the reference model and the algorithm for modular AC generation. This was implemented with the prototype SARMER tool and demonstrated in an illustrative example of a modular AC for a simple component-based architecture.

We plan to conduct several more thorough industry case studies in order to verify whether the SARM model is complete and efficient in use for large systems and product lines. The case studies will also help in identifying the need to extend the parameter and condition mechanisms for assurance case templates.

The technology used in the SARMER tool provides easy extension and integration with other systems. Our intention is to continue further development and extension of the tool to enable AC maintenance, provide an easy-to-use mechanism of model change management and automate data between different tools integrated in the system assurance process. Later, we plan to incorporate such functionality into the NOR-STA tool.

`

References

[1] International Organization for Standardization (ISO), “ISO 26262:2018 Road Vehicles - Functional Safety”, 2018.

[2] International Organization for Standardization (ISO), “ISO/SAE 21434:2021 Road vehicles — Cybersecurity engineering”, 2021.

[3] European Committee for Electrotechnical Standardization, “Railway applications - communication, signaling and processing systems - safety related electronic systems for signaling”, 2018.

[4] Civil Aviation Authority, “CAP 670: Air Traffic Services Safety Requirements”, The Third Issue, Amendment 1/2019, 2019.

[5] Underwriters Laboratories, “UL4600 - Standard for the Evaluation of Autonomous Products”, 2022.

[6] U.S. Food and Drug Administration, “Infusion Pumps Total Product Life Cycle Guidance for Industry and FDA Staff”, 2014.

[7] The Assurance Case Working Group, “Goal Structuring Notation Community Standard Version 3”, 2021.

[8] C. B. Weinstock, J. B. Goodenough and J. J. Hudak, “Dependability Cases”, CMU/SEI-2004-TN Technical Report, Software Engineering Institute, 2004.

[9] J. Cheng, M. Goodrum, R. Metoyer and J. Cleland-Huang, “How Do Practitioners Perceive Assurance Cases in Safety-Critical Software Systems?” In IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 57-60, IEEE, 2018.

[10] T. Myklebust, G. K. Hanssen and N. Lyngby, “A survey of the software and safety case development practice in the railway signalling sector” ESREL Portoroz, Slovenia, 2017.

[11] C. Almendra, C. Silva, L. Martins and J. Marques, “How assurance case development and requirements engineering interplay: a study with practitioners”, Requirements Engineering, 27(2), pp. 273-292, 2022.

[12] Y. Zhang, B. Larson and J. Hatcliff, “Assurance case considerations for interoperable medical systems”, In ASSURE 2018, International Conference on Computer Safety, Reliability, and Security, pp. 42-48, Springer, Cham, 2018.

[13] F. Warg, H. Blom, H., J. Borg and R. Johansson, “Continuous deployment for dependable systems with continuous assurance cases” In 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 318-325, IEEE, 2019.

[14] D. Nešić, M. Nyberg, M. and B. Gallina, “Product-line assurance cases from contract-based design”, Journal of Systems and Software, 176, 110922, 2021.

[15] P. Graydon, J. Knight and E. Strunk, “Assurance based development of critical systems”, In 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pp. 347-357, 2007.

[16] T. Myklebust and T. Stålhane, “The agile safety case”, pp. 1-213, Berlin: Springer, 2018.

[17] A. Salameh and O. Jaradat, “A safety-centric change management framework by tailoring agile and V-model processes”, In 36th International System Safety Conference ISSC 2018, Phoenix, United States, 2018.

[18] R. Johansson and P. Koopman, “Continuous Learning Approach to Safety Engineering”, In CARS-Critical Automotive applications: Robustness & Safety, 2022.

[19] M. Maksimov, N. Fung, S. Kokaly and M. Chechik, “Two decades of assurance case tools: a survey”, In ASSURE 2018, International Conference on Computer Safety, Reliability, and Security, pp. 49-59, Springer, Cham, 2018.

[20] E. Denney and G. Pai, “Tool support for assurance case development” Automated Software Engineering, Vol. 25, No. 3, pp. 1-65, 2017.

[21] M. Zeller, „Towards Continuous Safety Assessment in Context of DevOps”, In International Conference on Computer Safety, Reliability, and Security, pp. 145-157, Springer, Cham, 2021.

[22] T. Kelly, “Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems”, in Proceedings of Eleventh Australian Workshop on Safety-Critical Systems and Software, Melbourne, Australia, 2006.

[23] J. Fenn, R. Hawkins, T. Kelly and P. Williams, “Safety Case Composition Using Contracts – Refinements Based on Feedback from an Industrial Case Study”, 15th Safety Critical Systems Symposium, 2007.

[24] O. Jaradat, I. Bate and S. Punnekkat, “Facilitating the maintenance of safety cases”, In 3rd International Conference on Reliability, Safety and Hazard - Advances in Reliability, Maintenance and Safety, pp. 349–371, Springer, 2015.

[25] T. Kelly and J. McDermid, “Safety case patterns – reusing successful arguments”, Proceedings of IEE Colloquium on Understanding Patterns and Their Application to System Engineering, London, UK, 1998.

[26] E. Denney and G. Pai, “Safety Case Patterns: Theory and Applications”, NASA/TM–2015–218492 Technical Report, 2015.

[27] M. Szczygielska and A. Jarzębowicz, “Assurance Case Patterns On-line Catalogue” In: Advances in Dependability Engineering of Complex Systems (DepCoS-RELCOMEX 2017), Advances in Intelligent Systems and Computing vol. 582, pp. 407-417, Springer, Cham, 2017.

[28] L. Cyra and J. Górski, „SCF—A framework supporting achieving and assessing conformity with standards”, Computer Standards & Interfaces, 33(1), pp. 80-95, 2011.

[29] A. Wassyng, P. Joannou, M. Lawford, T. S. Maibaum, and N. K. Singh, “New Standards for Trustworthy Cyber-Physical Systems”, in Trustworthy Cyber-Physical Systems Engineering. CRC Press, pp. 337–368, 2016.

[30] R. Bloomfield, G. Fletcher, H. Khlaaf, L. Hinde and P. Ryan, “Safety case templates for autonomous systems”, arXiv preprint arXiv:2102.02625, 2021.

[31] R. Wei, T. Kelly, X. Dai, S. Zhao and R. Hawkins, “Model based system assurance using the structured assurance case metamodel”, Journal of Systems and Software, 154, pp. 211-233, 2019.

[32] E. Denney and G. Pai, “A lightweight methodology for safety case assembly”, In 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2012), pp. 1-12, Springer Berlin Heidelberg, 2012.

[33] A. Wardziński and A. Jarzębowicz, “Towards Safety Case Integration with Hazard Analysis for Medical Devices”, In Proc. of 4th International Workshop on Assurance Cases for Software-intensive Systems (ASSURE 2016), LNCS 9923, pp. 87-98, 2016.

[34] E. Denney and G. Pai, “Architecting a Safety Case for UAS Flight Operations”, Proceedings of 34th International System Safety Conference (ISSC 2016), 2016.

[35] A. Gacek, J. Backes, D. Cofer, K. Slind and M. Whalen, “Resolute: an assurance case language for architecture models” In ACM SIGAda Ada Letters Vol. 34, No. 3, pp. 19-28, 2014.

[36] M. Khalil, B. Schätz and S. Voss, “A Pattern-based Approach towards Modular Safety Analysis and Argumentation”. Embedded Real Time Software and Systems Conference (ERTS 2014) – Toulouse, France, February, 2014.

[37] Cârlan, C., Barner, S., Diewald, A., Tsalidis and S. Voss, “ExplicitCase: integrated model-based development of system and safety cases”, In International Conference on Computer Safety, Reliability, and Security, pp. 52-63, Springer, Cham, 2017.

[38] S. Ramakrishna, H. Jin, A. Dubey and A. Ramamurthy, “Automating Pattern Selection for Assurance Case Development for Cyber-Physical Systems”, In Computer Safety, Reliability, and Security: 41st International Conference, SAFECOMP 2022, pp. 82-96, Springer International Publishing, 2022.

[39] I. Šljivo, G. J. Uriagereka, S. Puri and B. Gallina, “Guiding assurance of architectural design patterns for critical applications”, Journal of Systems Architecture, Volume 110, 101765, 2020.

[40] B. Gallina, “A model-driven safety certification method for process compliance”, In 2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 204-209, 2014.

[41] R. Hawkins, T. Richardson and T. Kelly, “Using process models in system assurance” In: International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2016), Springer International Publishing, pp. 27-38, 2016.

`

[42] F. UL Muram, B. Gallina and L. Gómez Rodríguez, “Preventing Omission of Key Evidence Fallacy in Process-Based Argumentations”, 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 65-73, 2018.

[43] I. Sljivo, B. Gallina, J. Carlson and H. Hansson, “Generation of safety case argument-fragments from safety contracts” In International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2014), pp. 170-185, 2014.

[44] N. Basir, E. Denney and B. Fischer, “Deriving safety cases for hierarchical structure in model-based development”, Proc. of 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP ‘10), Vienna, Austria, 2010.

[45] Y. Matsuno and K. Taguchi, “Parameterised argument structure for GSN patterns”, In 11th International Conference on Quality Software (QSIC), pp. 96-101, IEEE, 2011.

[46] R. Hawkins, I. Habli, D. Kolovos, R. Paige and T. Kelly, “Weaving an Assurance Case from Design: A Model-Based Approach”, 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE), 2015.

[47] A. Wardziński and P. Jones, “Uniform model interface for assurance case integration with system models”, In: Proceedings of the 36th International Conference on Computer Safety, Reliability, and Security (SAFECOMP). vol. 10488, pp. 39–51, Springer, 2017.

[48] A. Retouniotis, Y. Papadopoulos, I. Sorokos, D. Parker, N. Matragkas and S. Sharvia, “Model-connected safety cases”, In International Symposium on Model-Based Safety and Assessment, pp. 50-63, Springer, Cham, 2017.

[49] S. Alajrami, B. Gallina, I. Sljivo, A. Romanovsky and P. Isberg, “Towards cloud-based enactment of safety-related processes”, In Proc. of 35th International Conference on Computer Safety, Reliability, and Security, pp. 309-321, 2016.

[50] A. Agrawal, S. Khoshmanesh, M. Vierhauser, M. Rahimi, J. Cleland-Huang and R. Lutz, “Leveraging artifact trees to evolve and reuse safety cases”, In IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1222-1233, IEEE, 2019.

[51] C. Cârlan, L. Gauerhof, B. Gallina and S. Burton, “Automating Safety Argument Change Impact Analysis for Machine Learning Components”, In 27th Pacific Rim International Symposium on Dependable Computing (PRDC 2022), pp. 43-53, IEEE, 2022.

[52] K. Attwood, T. Kelly and P. Conmy, “The use of controlled vocabularies and structured expressions in the assurance of CPS”, Ada User Journal, pp. 251-258, 2014.

[53] E. Denney and G. Pai, “A formal basis for safety case patterns”, In Computer Safety, Reliability, and Security: 32nd International Conference SAFECOMP 2013, pp. 21-32, Springer, 2013.

[54] P. Graydon, J. Knight and K. Wasson, “A flexible approach to authorization of UAS software”, In: IEEE/AIAA 28th Digital Avionics Systems Conference, pp. 5-C, 2009.

[55] SAE International, “SAE J3016: Levels of Driving Automation”, version 3, 2020.

[56] International Organization for Standardization (ISO), “ISO/FDIS 34503 Road Vehicles — Test scenarios for automated driving systems — Specification for operational design domain” (standard under development).

[57] Argevide NOR-STA homepage, https://www.argevide.com/assurance-case/

[58] Object Management Group, “Structured Assurance Case Metamodel”, ver. 2.3, 2022, https://www.omg.org/spec/SACM/About-SACM/